Master of Artificial Intelligence

Speeding up Reinforcement Learning with Learned Models

Bartomeu Pou Mulet

Supervisor: Mario Martin

Universitat Politecnica de Catalunya
Universitat de Barcelona
Universitat Rovira i Virgili

October 16, 2019

Acknowledgements

I would like to thank my master thesis supervisor Mario Martin. His enthusiasm in Reinforcement Learning inspired me to work on this topic and without his help, this thesis would not have been possible. It has been a blast discussing ideas of the Reinforcement Learning domain with you.

Also, I would like to thank my friends for their support during the development of this thesis. Manel, thanks to you I believe in myself. Jaume, thanks for being my mentor in the workforce. Gaston, thanks for helping me pushing my boundaries. Antonio, Gerard, Leonardo, Claudia and Sara you gave me good laughs and support when needed. I also would like to thank my friends from Mallorca who helped me become who am I now.

Finally, I would like to thank my family for the support I have received during my life and specially, these last months.

Abstract

Reinforcement Learning paradigm has obtained impressive results in solving complex tasks such as videogames [23] or robotic control [11]. Despite its success, two main problems hinder its applicability to many other problems: low sample efficiency (needing high amount of samples to learn to solve a problem) and sparse rewards (when most of the time reward signals are not informative enough to learn from).

Model Based Reinforcement Learning shows promise to solve the sample efficiency problem. This kind of algorithms use the data, not only to learn an optimal policy via trial and error, but to create a model that can be used to generate synthetic data. Then, traditional Model Free algorithms, can be used in combination with high quantities of synthetic data to increase sample efficiency.

Concerning Model Free methods, off-policy algorithms, those which do not need to follow the same policy which they are learning from, are more sample efficient than their counterparts: on-policy algorithms [11] [21]. The combination of Model Based Reinforcement Learning with off-policy methods has yet to be explored [20].

The sparse reward problem appears when the reward function usually returns the same reward except for the goal state and when the reward is hard to find by trial and error. One of the most notable contributions to solve this problem is Hindsight Experience Replay [1]. This method is able to learn from erroneous trajectories that do not reach the goal state by treating some states of experienced trajectories as goal states and thus receiving a reward signal which the agent can learn from.

In this master thesis, we explore the three mentioned topics: off-policy algorithms, Model Based Reinforcement Learning and Hindsight Experience Replay. For that purpose, we implement Hindsight Experience Replay with a recent state of the art off-policy algorithm Soft Actor Critic (SAC) [11]. Then, we extend a robust Model Based Reinforcement Learning method that works with on-policy methods, Model Ensemble Trust Region Policy Optimization (ME-TRPO) [19], to work with off-policy methods, which we call, Model Ensemble Soft Actor Critic (ME-SAC). Finally, we combine ME-SAC with Hindsight Experience Replay to solve sparse reward problems with more sample efficiency.

Results show that a) HER works perfectly with SAC, b) ME-SAC is able to learn from imagined data in a modified Half Cheetah environment and c) ME-SAC with HER is 10 times faster than its model free counterpart in Fetch and Reach environment.

List of Figures

List of Tables

Table of Contents

1 Introduction

In this master thesis, we try to tackle two of Reinforcement Learning most prominent problems: **sparse rewards** (when most of the time reward signals are not informative enough to learn from) and **low sample efficiency** (needing high amount of samples to learn to solve a problem). To do that we have proposed a combination of algorithms that are sample efficient (Model Based Reinforcement Learning and off-policy methods) and that work with sparse rewards (Hindsight Experience Replay).

1.1 Motivation

The central core our motivation is to make Reinforcement Learning more similar to human learning and thanks to this, to make it easier for it to be applied to real world problems.

While humans are able to learn to manipulate objects or to play a video game just by interacting a few times with a given problem, Reinforcement Learning algorithms take millions of interactions to be able to solve a problem optimally [23] [30]. An answer to this problem may be Model Based Reinforcement Learning, where a model is built to mimic the environment dynamics. Model Based Reinforcement Learning draws inspiration from humans as they have an internal model of their surroundings and can predict outcomes of their actions which then can be used in learning the correct policy.

Increasing sample efficiency can make a path for Reinforcement Learning to be applied extensively. For example, we can train in the physical world a robotic arm to grab objects by just feeding our agent visual input. However this problem may take millions or tens of millions of interactions with the object in order for our algorithm to learn the optimal policy and thus making it not viable due to time and energy expenditure. A possible solution could be increasing the sample efficiency with, for example, Model Based approaches, up to a point to make this kind of problems feasible.

Moreover, most of the tasks require complex handcrafted reward functions in order to speed up the learning process to a reasonable time. A desired feature of this kind of algorithms would be changing the complex reward functions, by simpler ones, such as giving a reward only if they reach the desired goal. In complex problems, using simple reward functions, may lead to an agent that never obtains a signal that it can learn from, as it never reaches its goal just by random walk. This kind of rewards are called sparse. Solving the sparse reward problem would contribute to make Reinforcement Learning succeed without human input (by not creating the complex reward functions) and thus making steps into General Artificial Intelligence.

The idea of this work is to test an integration of Model Based Reinforcement Learning with off-policy methods to increase sample efficiency and algorithms that work with sparse rewards.

1.2 Contributions

The two main contributions of this thesis have been twofold:

1. Extend a paper of Model Based Reinforcement Learning changing the on-policy method used (TRPO) for an off-policy one (SAC). As mentioned in [20], deep off-policy Model Based methods are not studied at all. To our knowledge, this thesis marks one of the first entries to study the combination of off-policy with Model Based methods.

2. Merge the Model Based Reinforcement Learning with off-policy methods with another method which works with sparse rewards (Hindsight Experience Replay). We show that the combination can obtain $10x$ faster results in the environment Fetch and Reach.

1.3 Overview

This work is divided into four different parts.

First, in Reinforcement Learning review (section 2), we will explain the theoretical background needed for this thesis. In section 2.1, we will give a brief introduction to the Reinforcement Learning problem. In sections 2.2 and 2.4, we will explain both classes of Reinforcement Learning algorithms we are using: Model Free and Model Based. In both 2.2 and 2.4, we will mention some of the latest papers that appeared in the field on those subjects. In section 2.3, a Model Free method to deal with sparse rewards will be introduced.

Next in section 3, we will explain how we have integrated the solution for sparse rewards and Model Based Reinforcement Learning. Specifically, in section 3.1, we will explain how we have reproduced the Model Based Reinforcement Learning solution changing on-policy to off-policy, in section 3.2, we will explain how we have merged the sparse rewards solution with the Model Based one and, in section 3.3, we will explain the domains in which we have tested our algorithms.

After that, in section 4, we will explain the results for each contribution separately. In section 4.1, we test test the method that works with sparse rewards with a Model Free off-policy state of the art algorithm. In 4.2, we investigate Model Based Reinforcement Learning with an off-policy method. In 4.3 we report the results of our objective: the final combination of Model Based Reinforcement Learning method with a method that works with sparse rewards.

Finally, in section 5, we will end this thesis with the conclusions of our work and with possible future work.

2 Reinforcement Learning review

This section will be focused on explaining the theoretical background of our thesis. Most of the conceptual basis and notation is extracted from the original work of Sutton & Barto [33].

2.1 The Reinforcement Learning problem

Reinforcement Learning tackles the problem of learning a policy via trial and error that maximises the long term expected reward.

The Reinforcement Learning problem is set in a world or **environment** where its characteristic are given by a multi-dimensional vector called **state,** s. An **agent** can observe part of those characteristics in a multi-dimensional vector called **observation,** o ($o \in s$)[1], it is able to act on it with an **action,** a, and after each action, the agent is presented with a scalar signal called **reward,** r, which measures its performance. The action affects the next state of the environment and reward and even the subsequent states and rewards. The rewards may be delayed i.e. the consequence of an action may not be seen until a period of time into the future.

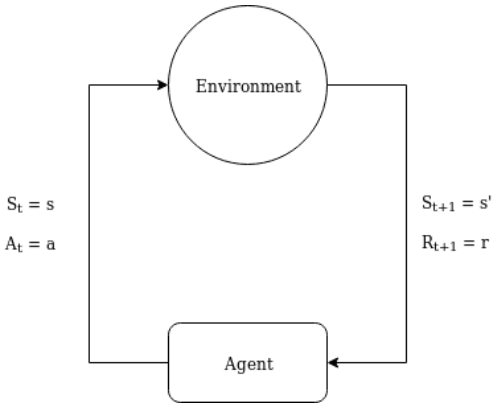


Figure 1: Reinforcement Learning Process. Given a state, s, and an action, a, the environment transitions to state s' and gives out a reward signal r.

Formally the Reinforcement Learning problem is defined as a **Markov Decision Process** formed by a set of states $\mathbb{S}$ ($s \in \mathbb{S}$), a set of actions $\mathbb{A}$ ($a \in \mathbb{A}$), a reward function $r(s, a) : S\ x\ A \rightarrow \mathbb{R}$ (which for notation we will use R_t indicating the output of the reward function at time t or r if it is the scalar value) and a probability transition function between states $p(s'|s, a) : S\ x\ S\ x\ A \rightarrow [0, 1]$.

A Markov Decision Process is such process that has the **Markov Property** i.e. the transition function only depends on the action and the state from the previous timestep.

[1]For simplicity, we will use the notation where the agent fully knows the characteristics of the environment $s = o$.

$$p(S_{t+1}|S_t, A_t, ..., S_0, A_0) = p(S_{t+1}|S_t, A_t) \tag{1}$$

The goal of Reinforcement Learning algorithms is to learn how to act in order to maximize the **return**, G_t or long term reward up to an **horizon**, T.

$$G_t = R_{t+1} + ... + R_{t+n-1} + R_{t+n=T} = \sum_{i=t+1}^{t+T} R_i \tag{2}$$

In our case, we will tackle problems where the Horizon is infinite. In order for the infinite horizon problems returns be finite, a parameter $\gamma \in [0, 1)$ is introduced.

$$G_t = R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + ... = \sum_{i=t+1}^{\infty} \gamma^{i-t-1} R_i \tag{3}$$

In addition to make the return finite, γ controls how much weight is given to future rewards. A lower γ will give more importance to closer timesteps into the future and viceversa.

To maximize the return, the agent should choose the actions that have the desired effect. A **policy**, $\pi(a|s) \propto \mathbb{P}(A_t = a|S_t = s)$ is how an agent chooses actions depending on the state it is in and it is represented as a conditional probability distribution. Policies can be or stochastic or deterministic and discrete or continuous depending on the method you use. One way to find the optimal policy is introducing the concept of value functions[2]:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \middle| S_t = s \right] \tag{4}$$

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \middle| S_t = s, A_t = a \right] \tag{5}$$

Equation (4) is the **state value function**, $v_\pi(s)$, which is the expected return you would receive if you start in state s and follow the policy π afterwards. Equation (5) is the **state action value function** or **Q-value**, $q_\pi(s, a)$, which is the expected return you would receive if you start in state s, take action a and follow policy π afterwards. With these two concepts, we can quantify how good is staying in the states and how good is taking actions in certain states and then, we can quantify how good a policy is. A policy, π, is said to be better than another policy, π', if and only if $v_\pi(s) \geq v_{\pi'}(s)$.

In addition, we can expand both $v(s)$ and $q(s, a)$ in order for us to be able to estimate their value [33]:

[2]The expectation of $\mathbb{E}_\pi$ is over the transition probabilities $p(S_{t+1}|S_t, A_t)$. The π under the expectation is a convention that means that the actions are sampled according to the policy after the starting state, s, in the case of the state value function and after the starting state and starting action, s, a, in the case of the action state value function.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t \middle| S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... \middle| S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + ...) \middle| S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \middle| S_t = s \right] \\ &= \mathbb{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) \middle| S_t = s] \end{aligned} \tag{6}$$

This expression is called **Bellman equation for state value function** and we can use it to estimate $v_\pi(s)$ iteratively. The process consists in repeating in iterations, i, the Bellman equation $\forall s$: $v_\pi^{i+1}(s) = \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi^i(S_{t+1}) \middle| S_t = s \right]$ which converges for $i \to \infty$. This is called **policy evaluation** because you calculate the return of acting according to a certain policy. Practically, the number of iterations is reduced to 1 because we do not need the converged value of $v_\pi(s)$ to compare with other policies and due to the excessive computing time if we want to have a converged $v_\pi(s)$. The expression for $q_\pi(s, a)$ can be estimated similarly:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi \left[G_t \middle| S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \middle| S_t = s, A_t = a \right] \end{aligned} \tag{7}$$

This is the **Bellman equation for action state value function**. A relation between $v_\pi(s)$ with $q_\pi(s, a)$ can be found sampling the action in $q_\pi(s, a)$ from the policy:

$$v_\pi(s) = \mathbb{E}_{a \sim \pi} \left[q_\pi(s, a) \right] \tag{8}$$

Consider a deterministic policy π, if we estimate $q_\pi(s, a)$ we will know if taking a different action not recommended by the policy is better. If $q_\pi(s, a) \geq v_\pi(s)$ then if we consider a new policy π' that follows a in s, $\pi' \geq \pi$. This is called **policy improvement theorem**. From this, we can improve the policy and obtain an improved state value function choosing the action that maximises $q_\pi(s, a)$.

$$\pi'(s) = \arg\max_a q_\pi(s, a) \tag{9}$$

$$v_{\pi'}(s) = \max_a q_\pi(s, a) \tag{10}$$

The process of choosing a policy that is better than the current one is called **policy improvement**. The Reinforcement Learning problem can be summarised as finding the optimal policy (which we will mark with an asterisk, *):

$$\pi_*(s) = \arg\max_\pi v_\pi(s) \tag{11}$$

$$\pi_*(s,a) = \arg\max_{\pi} q_\pi(s,a) \tag{12}$$

The optimal policy achieves optimal state value function and state action value function, hence, finding optimal value functions is also solving the Reinforcement Learning problem. Both optimal value functions are given by:

$$v_*(s) = \max_{\pi} v_\pi(s) \tag{13}$$

$$q_*(s,a) = \max_{\pi} q_\pi(s,a) \tag{14}$$

In addition we can reformulate equation (10) to obtain the expression for optimal value functions.

$$v_*(s) = \max_{a} q_{\pi_*}(s,a) \tag{15}$$

Finally given the last expression, equation (15), we can rewrite the Bellman equations for optimal value functions with the name **Bellman optimality equations**:

$$v_*(s) = \max_{a} \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \middle| S_t = s, A_t = a\right] \tag{16}$$

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \middle| S_t = s, A_t = a\right] \tag{17}$$

To estimate either $v_*(s)$, $q_*(s,a)$ or $\pi_*(s)$ a two step procedure is used:

1. **Policy evaluation**: estimate the value functions for the current policy using the Bellman equations (6) (7).
2. **Policy improvement**: change the policy for a better one using the max function as in equation (9).

The two step procedure is called **policy iteration** (if the number of evaluations from the Bellman equation tends to ∞) or **value iteration** (if we just use one update for evaluation).

Value and policy iteration belongs to a family of methods called **Dynamic Programming**. The two main weaknesses of Dynamic Programming is that we need to know the dynamics of the system, $p(S_{t+1}|S_t, A_t)$, to compute expectations and that it suffers from **Curse of Dimensionality** because the number of possible states action pairs scale exponentially with the number of variables that define states and actions [33]. Reinforcement Learning estimates the values from experience, thus, it is able to solve these problems without the transition probabilities.

Before we start to dive into the subject we need to make a distinction of two families of Reinforcement Learning algorithms:

1. **Model Free Reinforcement Learning** where you estimate a value function or policy from experience.

2. **Model Based Reinforcement Learning** where you estimate the dynamics of the system from experience and use them to obtain the desired policy.

The full classification of the different families of Reinforcement Learning can be found below:

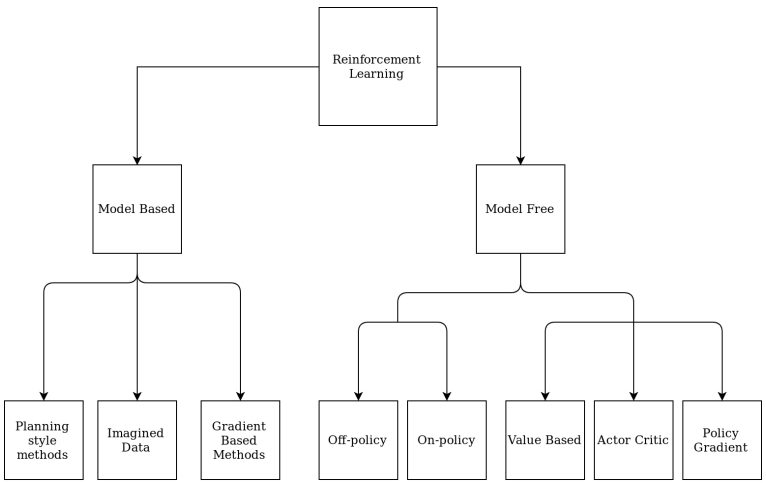


Figure 2: Reinforcement Learning classification. Both Model Based and Model Free have their own subclassifications which are explained in this chapter.

2.2 Model Free Reinforcement Learning

Model free methods are the families of methods that do not know the environment transition probabilities beforehand nor they try to estimate it.

They are divided depending on how the optimal policy is obtained:

1. **Value function methods**: which use $v_\pi(s)$ and $q_\pi(s, a)$ in order to estimate the optimal policy.

2. **Policy gradient**: which estimate the optimal policy using the definition of return as a cost function and gradient ascend to find it.

3. **Actor Critic methods**: which is a combination of both.

2.2.1 Value function methods

Consider a tuple formed by the states, actions, rewards and next states taken in an environment, what is known by a **trajectory**, $\tau = (S_0, A_0, S_1, R_1, ..., S_{T-1}, A_{T-1}, S_T, R_T)$. As explained, Reinforcement Learning uses value iteration (composed with policy prediction and policy improvement) to solve the problem:

1. **Prediction**: the prediction of the value of a state takes the form:

$$v(S_t) = v(S_t) + \alpha(f(\tau) - v(S_t)) \tag{18}$$

If $f(\tau) = G(\tau)$ (where $G(\tau)$ is the return from trajectory τ) we will be minimising the difference between $v(S_t)$ and long term return and it will eventually converge to the real $G(\tau)$. These are called **Monte Carlo (MC)** methods and use full trajectories to obtain the value of $v(S_t)$.

Instead if $f(\tau) = R_{t+1} + v(S_{t+1})$ (where R_{t+1} and $v(S_{t+1})$ come from one step from the trajectory) and we update the value function after each timestep we will be minimising what is called **Temporal Difference error** $R_{t+1} + v(S_{t+1}) - v(S_t)$. The prediction is updated after each action (i.e. **online**) and is based on the value function of the timestep before. This family of methods are called **Temporal Difference (TD)** methods.

An intermediate step between TD and MC are the **TD(n)** methods which take into account n steps into the future instead of one step or the full trajectory. In our case we will focus on TD(0), as we want an online algorithm which can be used with ease.

Both Temporal Difference learning and Monte Carlo are proved to converge to v_π [33].

2. **Improvement**: The policy is updated taking into account equation (9): you choose a new policy that maximizes the Q-values for all states. The main problem of always **greedily** choosing a new policy (i.e. choosing the policy that holds the maximum Q-values which is called **exploiting**) is that if many of state-action pairs are not visited the agent may end in a local minima because it does not know the rewards received of taking actions in those unknown states and will just go to the best known states.

That is why sometimes the agent may take an action that is not the maximum of the current Q-values in order to visit unexplored situations, also known as **exploration**. For example, for discrete policies ϵ-greedy exploration technique can be used:

$$\pi(a|s) = \begin{cases} \arg\max_{a'} q(s, a'), & \text{if } \epsilon < \text{random}(0, 1) \\ \text{random action}, & \text{otherwise} \end{cases} \tag{19}$$

The question of, for a given state is better to explore or follow the greedy option is called **exploitation-exploration dilemma** and is one of the main problems of Reinforcement Learning.

An important distinction of model free algorithms are the ones that update taking into account the policy that they are following, called **on-policy**, or the ones that can update taking into account other policies that are not their own, called **off-policy**.

One of the most used TD(0) off-policy algorithms is Q-learning [35]. Q-learning update rule is based on estimating directly $q_*(s, a)$ as shown in equation **(17)**:

$$q(s, a) = q(s, a) + \alpha(r + \gamma \max_{a'}(q(s', a')) - q(s, a)) \tag{20}$$

Where s is the state from the last time step, s' is the obtained state after taking an action, a, using the current policy and a' is selected using the max over the Q-values. As the action, a', selected for the update is the one that maximizes the Q-value, instead of the action selected by the current policy (that includes exploratory actions), we say that it is an off-policy algorithm.

The parameter α is the learning rate ($0 \leq \alpha \leq 1$) which controls how fast the Q-values change from one update to another. It has been shown that the values of α need the following properties for convergence: $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$ but in practice a constant α is used.

Up to this point we have considered Q-values represented as a table. As the number of states and actions increase, the Curse of dimensionality starts to affect the performance of traditional Reinforcement Learning methods. To visit all of the states and actions in a tractable time and store the Q-values into the memory becomes impossible. That is the reason that a function that approximates $v_\pi(s)$ and/or $q_\pi(s, a)$ is used. Consider a function $\widehat{q}(s, a, w)$ with a weight vector w that approximates the value of $q(s, a)$.

$$\widehat{q}(s, a, w) \simeq q(s, a) \tag{21}$$

The function approximator can take many forms such as a simple linear regression or decision trees but in our case we will focus on neural networks as recent work in Reinforcement Learning has shown its powerful usage [23] [30].

With neural networks the natural extension of Q-learning is **Deep Q-learning (DQN)** presented in [23]. In the case of DQN, the neural network will be trained in a supervised way trying to minimise the difference between the prediction of the neural network and the true value of the Q-value.

$$L(w) = (y_i - \widehat{q}(s, a, w))^2 \tag{22}$$

Where y_i is the target, the real value of $q_\pi(s, a)$ with parameters in update i, that is estimated using Bellman equation (same target as used in traditional Q-learning):

$$y_i = r + \gamma \max_{a'}(\widehat{q}(s', a', w_{i-1})) \tag{23}$$

A few tricks are used for DQN be able to obtain an optimal value and converge faster.

- **Replay buffer,** $\mathcal{D}$. A memory buffer is used to store recently experienced tuple sequences of state, action, reward, next state (i.e. (s, a, r, s')). The memory buffer was first introduced in [22] as replay buffer. At training time they are sampled randomly in mini-batches of small size.

$$L(w) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[(y_i(w) - \widehat{q}(s, a, w))^2\right] \tag{24}$$

  The act of training the neural network with samples from the replay buffer is called **experience replay**. As Mnih et al. in [23] puts, this has two advantages. First, a tuple (s, a, r, s') can be used more than once as the samples are randomly sampled hence more data efficiency. Second, there is a high correlation between consecutive steps in the environment as they will be taken from similar points in the state action space. Sampling randomly implies that two tuples in the same mini-batch will probably be far apart in the action state space breaking the correlation thus making the neural network more robust.

  Algorithms that use replay buffer are off-policy because samples from previous iterations of the current policy may be used for the update.

- **Target network** with weights w'. In order to not make the target, $y_i(w)$, dependant on the same parameters as $q(s, a, w)$, a separate network is used to compute the loss and every x steps w' and w are copied or averaged. This has an advantage that the target does not change after every update making the learning more stable. The equation for the loss becomes:

$$L(w, w') = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[(y_i(w') - \widehat{q}(s, a, w))^2\right] \tag{25}$$

2.2.2 Policy Gradient methods

Policy gradient methods is a family of algorithms that represent the policy as a probability distribution defined by some weights, w, $\pi(a|s, w) = \mathbb{P}_w[a|s, w]$. These methods take into account the maximization of the objective of Reinforcement Learning, the return:

$$J(w) = \mathbb{E}_\pi(G_t|w) \tag{26}$$

Instead of using value functions, policy gradient methods directly increase the probability of actions that give more long term reward in the given state. The maximization of such objective can be done with gradient ascend on the weights, w i.e:

$$w = w + \alpha\nabla_w J(w) \tag{27}$$

Where α is the learning rate as usual of such optimization methods and $\nabla_w J(w)$ is the gradient of the objective function w.r.t. the weights, w. Sutton et al. [33] proved an expression for $\nabla J(w)$ for stochastic policies in what is called **policy gradient theorem**:

$$\nabla_w J(w) = \frac{1}{C}\sum_s \mu(s)\sum_a \left[q_\pi(s, a)\nabla\pi(a|s, w)\right] \tag{28}$$

Where $\mu(s)$ is the time spent visiting each state, s, called **on-policy distribution** and C is the length of a trajectory we consider to do the gradient . If states (s), and actions (a), follow a trajectory $\tau \sim (S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, ...)$ taken by the policy and considering that $q_\pi(s,a) = [G_t|S_t, A_t]$ we can use an expectation to obtain:

$$\nabla_w J(w) = \mathbb{E}_{\tau\sim\pi}\left[G_t \nabla \pi(A_t|S_t, w)\right] \tag{29}$$

REINFORCE [36] was one of the first policy gradient algorithms and used this expression to calculate the gradients of the objective function from sample trajectories. This expression has the policy gradients main idea: if the return is high the action probabilities from this trajectory will be increased. In order to make it more stable, instead of taking the return directly, we can use a difference between the return and a baseline, b. REINFORCE with baseline considers a baseline that is the mean of all returns from all trajectories seen so far:

$$\nabla_w J(w) = \mathbb{E}_{\tau\sim\pi}[(G_t - b)\nabla \pi(A_t|S_t, w)] \tag{30}$$

While REINFORCE works well in some domains, there is a problem with assigning reward to trajectories: you do not know which actions in those trajectories are the ones that make them outperform (a problem of credit assignment).

2.2.3 Actor critic methods

Following the policy gradient theorem, many algorithms include a baseline $b(s,a)$ that may depend on the state and/or action. This baseline can be substituted by value functions or action value functions and even be parametrized. Actor Critic algorithms have the same idea than policy gradient methods: they have a set of weights w_1 that parametrize the policy $\pi(w_1)$ (in what is called the **actor**) but also use another set of parameters w_2 to approximate $q_\pi(s,a)$ (in what is called the **critic**).

The most common baseline used by actor critic methods is the advantage function: $Adv_\pi(s,a) = q_\pi(s,a) - v_\pi(s)$ which compares the expected value following the policy against the expected value of taking an action (that can be different from the one selected by the policy) and then following the policy.

Two of the most common methods for on-policy actor critic are **Trust Region Policy Optimization (TRPO)** [28] and **Proximal Policy Optimization (PPO)** [29] that set constraints when updating weights in order to not change drastically the policy after each update hence making a more stable method. While this kind of algorithms usually achieve the optimal behaviour they need many iterations to arrive to it.

An interesting actor critic algorithm that follows a different path is **Deep Deterministic Policy Gradient**. In [31] Silver et al. proved a version of policy gradient theorem for deterministic policies and introduced an algorithm **Deterministic Policy Gradient** (DPG) which could work in an off-policy way. Lillicrap et al. introduced in [21], a modified version of DPG with neural networks as function approximators inspired in

the success of DQN for discrete action spaces. This modified version was called **Deep Deterministic Policy Gradient (DDPG)**.

DDPG learns a policy that is deterministic using experience replay becoming an off-policy algorithm. As it learns a deterministic policy, exploration is encouraged adding noise while training. Additionally, the update of the target network is produced with a weighted average between the last weights of the target and the original network, called polyak averaging with parameter ρ ($0 \leq \rho \leq 1$). ρ close to 1 adds more weight to the target network while close to 0 adds more weight to the online network.

When a batch is sampled from the experience replay, two equations are used to update the actor and the critic. The equation for the critic is similar to the one for DQN as in equation **(22)** except a slight difference, in value iteration the max function is used to obtain better policies. In the case of continuous action spaces, max is too costly to compute, so instead, the value computed from the target actor is used, $\pi(a', w'_\pi)$. The actor works as a approximate maximum because it is trained to maximise $q(s, a)$. The final equation is:

$$L(w) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[(y_i - \widehat{q}(s, a, w_q))^2\right] \tag{31}$$

$$\text{where} \quad y_i = r + (q(s', \pi(a', w'_\pi), w'_q)$$

The gradient showed by Silver et al. [31] is used to update the actor. Consider w'_q and w_q for the target and online networks respectively for the Q-values and w'_π and w_π for the policy network, the deterministic policy gradient is:

$$\nabla_{w_\pi} J \simeq \sum_i \nabla_a \widehat{q}(s, a, w_q)\Big|_{s=s_i, a=\pi(s_i)} \nabla_{w_\pi} \pi(s, w_\pi)\Big|_{s=s_i} \tag{32}$$

Which is used to update the actor with gradient ascend. Each update of DDPG can be seen as the general rule for value evaluation: prediction (when we update the critic) and improvement (when we update the actor).

A modified version of DDPG was introduced with **Twin Delayed Deep Deterministic Policy Gradient (TD3)** [9]. It uses three modifications in order to outperform the original DDPG.

1. A recurrent problem from Q-values is that they are usually overestimated [9]. For that, two networks that only differ in weight initialization and the batch sampling, are used to estimate the Q-values. When the Q-values are needed to compute the loss, the minimum of both networks is used. This is why it has the word *twin* on the name.

2. The policy is updated at half the speed than the Q-values. This is why it has the word *delayed* on the name.

3. Noise is added into the actions to calculate the target Q-value.

While DDPG (and TD3) is regarded as one of the best model free algorithms for continuous action spaces, it extremely depends on initialization of weights, it can end in local minima and you have to massively tune the hyperparameters to obtain the optimal behaviour.

2.2.4 Maximum Entropy Reinforcement Learning and Soft Actor Critic

Maximum Entropy Reinforcement Learning modifies the Reinforcement Learning framework adding an entropy term to the problem. In maximum entropy Reinforcement Learning, we are interested in calculating the entropy of the policy itself which is given by the following expression:

$$H(\pi(\cdot|s)) = -\sum_{a\sim\pi} P(a) log P(a) = -\,\mathbb{E}_{a\sim\pi}\left[log P(a)\right] \tag{33}$$

Equation **(33)** achieves maximum value if the policy follows an uniform distribution (i.e. random policy). Maximum entropy Reinforcement Learning modifies the original objective in order to introduce the entropy, becoming a trade off between return and maximum entropy:

$$J^{MaxEnt}(w) = \mathbb{E}_{(s,a)\sim\pi}\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} + \beta H(\pi(\cdot|s))\right] \tag{34}$$

β is called temperature parameter ($0 \leq \beta \leq 1$) and determines the strength of the entropy on the Reinforcement Learning problem. Value functions are changed to take into account the entropy and are called soft value functions:

$$q_\pi^{soft}(s,a) = \mathbb{E}_{(S_{i+1},A_{i+1},\ldots)\sim\pi}\left[R_{t+1} + \sum_{i=t+1}^{\infty} \gamma^i (R_{i+1} - \beta \log \pi(A_i|S_i)) \Big| S_i = s, A_i = a\right] \tag{35}$$

$$\begin{aligned} v_\pi^{soft}(s) &= \mathbb{E}_{(S_{i+1},\ldots)\sim\pi}\left[\sum_{i=t}^{\infty} \gamma^i \left[R_{i+1} - \beta \log \pi(A_i|S_i)\right] \Big| S_i = s\right] \\ &= \mathbb{E}_{a\sim\pi}\left[q_\pi^{soft}(S_i,a) - \beta \log \pi(a|S_i) \Big| S_i = s\right] \end{aligned} \tag{36}$$

As the objective now has an entropy term, the policies created with such have increased exploration capabilities and make algorithms more robust to the change of hyperparameters.

Soft Actor Critic [11] uses neural to approximate the soft value functions $\widehat{q}_\pi^{soft}(s,a,w_{1,1})$ and $\widehat{v}_\pi^{soft}(s,w_2)$ and the policy $\pi(a|s,w_3)$. An additional neural network for $\widehat{q}_{soft,\pi}(s,a,w_{1,2})$ is used in order to achieve faster convergence based on the trick of Twin Delayed Deep

Deterministic Policy Gradient (TD3). Additionally, as seen in equation (36), having the soft Q-value let's you compute the soft state value but the authors obtain increased stability if it is approximated with a neural network.

The learning is done off-policy sampling from an experience replay. The approximation of value function use target networks (denoted by the symbol $'$) and are updated using polyak average. The Q-values are learned minimizing the error of what the neural network is predicting to what the soft Bellman equation (which is the Bellman equation for soft value functions) for soft Q-values computes i.e:

$$L(w_{1,k}) = \mathbb{E}_{D\sim(s,a,r,s')}\left[(y_i - \widehat{q}_{soft}(s,a,w_{1,k}))^2\right] \tag{37}$$

$$y_i = r + \gamma \widehat{v}_{soft}(s', w_2')$$

The soft state value is computed comparing $\widehat{v}_{\pi}^{soft}(s, w_2)$ and what it should learn from equation (36). The expectation from the right hand side of the equation is removed and instead an action is sampled from the policy. As with TD3 the estimation of $\widehat{q}_{soft}$ is calculated from two different networks and the minimum of both is selected.

$$L(w_2) = \mathbb{E}_{D\sim(s,a,r,s')}\left[(y_i - \widehat{v}_{soft}(s,a,w_2))^2\right] \tag{38}$$

$$y_i = \min_k \widehat{q}_{soft}(s,a',w_{1,k}) - \alpha log\pi(a'|s),\ \ a' \sim \pi(a|s,w_3)$$

Finally, the policy is learned by taking the actions that maximizes the long term return (and in case of Maximum Entropy Reinforcement Learning, the entropy) i.e. $\widehat{v}_{\pi}^{soft}(s, w_2)$. To do that we can use the expression in (36) but we run into a problem, we need to sample an action from the policy while at the same time we try to optimize it. To solve this, **reparametrization trick** is used: actions are sampled with a deterministic function dependant on the state plus some noise (following a distribution $\mathcal{N}$), $a \sim f(s,\xi)$. Then, we can make the expression found on (36) not dependant on the actions of the policy which we are trying to update.

$$\mathbb{E}_{a\sim\pi}\left[q_{\pi}^{soft}(s,a) - \beta log\pi(a|s)\right] \sim \mathbb{E}_{\xi\sim\mathcal{N}}[q_{\pi}^{soft}(s, f(s,\xi)) - \beta log\pi(f(s,\xi))] \tag{39}$$

Then the final loss:

$$L(w_3) = \mathbb{E}_{\xi\sim\mathcal{N}}\left[min_k q_{\pi}^{soft}(s, f(s,\xi), w_{1,k}) - \beta log\pi(f(s,\xi))\right] \tag{40}$$

Soft actor critic improved objective led to a new state of the art for model free Reinforcement Learning. Many robotics environments where tested and it surpassed previous state of the art algorithms such as DDPG or TD3 in most of them. The full pseudocode can be found below:

Algorithm 1 Soft Actor Critic [11]

Initialize weights for q_{soft}, v_{soft} and π: $w_{1,1}$, $w_{1,2}$, w_2, w_3
Initialize replay memory D and target networks
for each iteration **do**
    Initialize s from the environment
    for step **do**:
        Choose a according to the policy: $a \sim \pi(a|s, w_3)$
        Execute a in the environment and observe s' and r.
        Append the tuple (s,a,s',r) in the replay memory, D.
    for number of updates required **do**
        $w_1 = w_1 - \nabla J(w_1)$
        $w_{2,k} = w_{2,k} - \nabla J(w_{2,k})$ for k = 1,2
        $w_3 = w_3 - \nabla J(w_3)$

2.2.5 Model Free problems: sample efficiency and sparse rewards

Model Free Reinforcement Learning tasks **require a high number of samples to successfully solve complex tasks**. For example, the solution of DQN for Atari games required millions of interactions with the environment to achieve optimal results [23]. Achieving less sample complexity will help bring practical applications such as robotics to the real world and make the learning more similar to the human learning process.

On-policy and Off-policy methods have different sample efficiency. On-policy algorithms are said to have poor sample efficiency because, after an interaction with the environment is used to updated the policy, it is discarded. Instead, off-policy algorithms can use more than once a same sample as they are sampled from the memory replay randomly at training time. Not only that but it has been shown time after time that off-policy algorithms perform better than on-policy algorithms in that they require less samples to obtain the optimal result [21] [23] [11]. Despite this off-policy algorithms are less stable than their counterparts, they may get very different results depending on the seed (because they end in an local minimum or because they diverge) and they require extensive hyperparameter search to work well. Soft actor critic [11] is a model free algorithm which tackles their weaknesses, trying to solve the stability problem of off-policy algorithms with the maximum entropy framework.

- **On-policy**: less sample efficient, more stable, less hyperparameter search.
- **Off-policy**: more sample efficient, less stable, more hyperparameter search.

The problem of sample efficiency is addressed by Model Based Reinforcement Learning using the samples even more than in off-policy Model Free Reinforcement Learning.

Many reinforcement learning problems assign to an agent meaningful reward after each action. For example consider the famous example of pendulum environment [25]. The problem consists in a pendulum and an agent that can exert a force to make it stay

upright. At each timestep, the agent receives a reward depending on the angle distance between his actual point to the objective point (being upright), which then it can use to learn the optimal policy. Instead the same problem could be formulated giving a reward of 1 if it arrives to the objective and 0 otherwise. In complex tasks this can make the Reinforcement Learning agent not to learn anything. The agent may not be able to arrive to the objective without modifying the initial policy and as it has not received any reward different than 0, it is not able to modify it. The rewards that are almost never meaningful are called **sparse rewards**.

The problem of sparse rewards will be addressed by Hindsight Experience Replay.

2.3 Hindsight Experience Replay

Hindsight Experience Replay (HER) [1] tries to address one of the major problems in Reinforcement Learning, sparse rewards. To explain HER we first need to talk about universal value function approximators [27].

Universal value function $V(s, g)$ are a modified version of value functions which indicate the expected reward of being in state s and having goal g. In other words, how good is the expected return depending on the goal the agent has. This can be useful in problems where a different set of goals have to be solved. Universal value function can be further expanded with function approximation. **Universal value function approximators (UVFA)** use an approximator such a neural network to parametrize the Q and V functions in the goal and state space i.e.

$$V(s, g) \simeq \hat{V}(s, g) \tag{41}$$

HER takes the idea of UVFA and uses it to learn in a problem with sparse rewards.

Consider an environment with a) a desired goal g_d, b) sparse rewards i.e. if $d(s, g_d) < \epsilon$ $r = 1$ else $r = 0$ (where $d(a, b)$ is distance between a and b and ϵ is an scalar close to 0 to accept a small error) and c) to achieve the goal is not trivial. In most of these kind of problems the agent is not able to solve the environment (arriving to g_d) while taking random actions and hence, the reward received will be the same for all timesteps meaning that there will be no meaningful signal from which it can learn.

HER works by extending the idea of experience replay. For every episode the usual state, action, reward, next state are appended to the memory buffer, but with one difference: the state and goal are concatenated i.e. $(s||g_d, a, r, s'||g_d)$ is appended to the replay buffer where $s||g_d$ denotes the concatenation of the state with the current goal. Then a set of additional goals are sampled (g') according to a strategy $\mathbb{S}$ and rewards are computed as if they were the original goals (r_g), then those new transitions $(s||g', a, r_g, s'||g')$ are appended to the replay memory. Thanks to this the agent is able to see meaningful rewards for every episode as it will always reach the synthetic goal.

With this method, impressively nonetheless, the agent is able to generalise between goals and according to [1] it can achieve the original goal even if it has never seen it. As HER only works with replay memory, it only will work with off-policy algorithms. To sample the additional goals, g', different sampling strategies, $\mathbb{S}$, are considered.

- **Final**: there is only one sampled goal and is the one where the episode ends.
- **Random**: the goals are randomly sampled across all the visited states over all training time.
- **Episode**: the goals are randomly sampled across all visited states in that episode.

- **Future**: the goals are randomly sampled across all visited states in that episode that come after current one.

All of these sampling strategies were tested on robotic tasks with sparse rewards on the physics simulator MUJOCO [34]. The chosen off-policy algorithm to test HER is DDPG. While DDPG is not able to achieve the objective after many iterations the combination of DDPG with HER is. In addition among all the sampling strategies, episode and future are the ones that reported better results. The full pseudocode can be found below:

Algorithm 2 Hindsight Experience Replay [1]

Given: an off-policy RL algorithm $\mathbb{A}$, a strategy for sampling goals $\mathbb{S}$, and a reward function dependant on goals $r(s, a, g)$
Initialize replay memory D
Initialize $\mathbb{A}$
for each episode **do**
    Initialize s from the environment and desired goal g_d
    for each step in the episode **do**:
        Choose a from $\mathbb{A}$ $a = \pi(s||g_d)$
        Execute action in environment and observe s'
        $s = s'$
    for each step taken in the episode **do**:
        $r = r(s_t, a_t, g_d)$
        Store tuple $(s||g_d, a_t, r_t, s_{t+1}||g_d)$ in replay memory, R
        Sample a set of additional goals from $\mathbb{S}$ for replay G.
        for $g' \in G$ **do**:
            $r' = r(s_t, a_t, g')$
            Store tuple $(s||g', a_t, r_t, s_{t+1}||g')$ in replay memory, R
    for for update = 1, N **do**
        sample a minibatch from the replay buffer
        Perform a gradient descent step on the minibatch

2.4 Model Based Reinforcement Learning

Model based Reinforcement learning focuses on learning a function $f(s,a) : S\ x\ A \rightarrow S\ x\ \mathbb{R}$, which models the environment dynamics. The function is an estimation of the next state, s', and in some cases the reward[3], r, given the initial state, s, and action, a: $f(s,a) \simeq (s', r')$. The learned dynamics only predict an state from the previous state as we are working with Markov Decision Processes, although in some cases such as in [10] additional previous states are used for prediction.

The dynamics are learned with supervised learning procedures, a dataset of interactions with the environment is initially obtained (s, a, r, s'), and then with a function approximator such as a neural network, $f(s,a)$ is learned. The learned function is sometimes called **virtual environment** or **world model** as it mimics the real environment approximately.

Three areas of Model Based Reinforcement Learning can be seen across the literature[4].

1. **Planning style methods** where given a state s, you sample different trajectories from that state into the future and select the actions that give maximum return.
2. **Gradient based methods**. In gradient based methods the model is used as an approximation to real dynamics to calculate the true gradients from the Reinforcement Learning objective.
3. **Imagined data methods**. In this kind of methods you interact with the world model as if it were a real model and obtain more data to update the policy. In our case we will focus on this kind of methods.

2.4.1 Planning style methods

Planning is an online algorithm, given a learned model $f(s,a)$ and an state s, you compute N trajectories, τ, up to a wanted horizon T, and obtain the given return G. After finishing you select the actions with $\tau = \arg\max_\tau G(\tau)$

[3]Depending on the problem you will want to learn the reward in addition to the next state but in our case we will focus on only the state.

[4]Sutton and Barto [33] classifies Model Based Reinforcement Learning (which they call planning) into background planning (where gradient and imagined data would be) and decision-time (which is the one that we call planning style methods). Recent literature does not use this terminology and we have adopted such [7] [20].

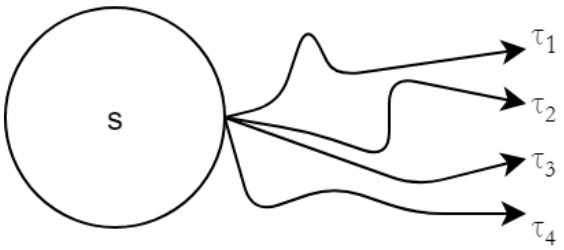


Figure 3: Planning methods.

Usually in order to obtain better results, only the first action of the chosen trajectory is chosen and for the following state the trajectory sampling is computed again. When only the first action is chosen this kind of algorithms are usually called **Model Predictive Control**.

A recent example of this family of methods can be seen in [24] where Nagabandi et al. used Model Predictive Control to sample optimal actions from imagined trajectories which then were used via imitation learning to initialize a policy for a Model Free method. Another example can be found in **Probabilistic Ensemble with Trajectory Sampling (PETS)** [4] where they use a collection of probabilistic neural networks to model the environment's stochasticity and then sample the desired trajectories.

In visual tasks, planning methods had a big success with **PlaNet (deep planning network)** algorithm [12]. This algorithm encodes the visual space into a lower dimensional space and then plans on it leading to impressive results.

2.4.2 Gradient based methods

Gradient based methods use the model dynamics to compute the gradient over long horizons **analytically** with techniques such as backpropagation and use them to improve the policy. Consider an imagined trajectory, $\tau = (s_0, a_0, r_0, ..., s_T, a_T, r_T)$. After finishing the trajectory and receiving a reward signal the gradient of the Reinforcement Learning objective function $J(\theta)$ with respect to θ can be computed thanks to the Bellman equation (6). In order to obtain this gradient analytically you need to obtain the transition probabilities. Those transition probabilities are estimated using the model.

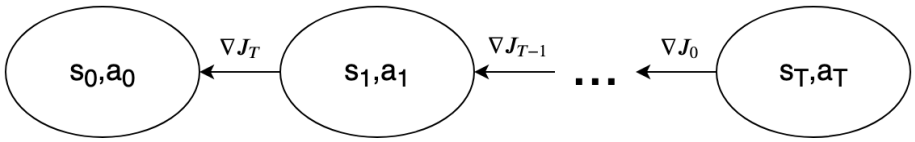


Figure 4: Gradient based methods. Over an imagined trajectory they are able to compute the gradient analytically.

An example of gradient based methods is **PILCO (probabilistic inference for learning and control)** which use Gaussian processes to build the model with which an analytical backpropagation can be computed. The main issue from this research is that Gaussian processes do not work well with high dimensional action state spaces.

Another example can be found in [13]. **Stochastic Value Gradients (SVG)** which, as PILCO, learn stochastic dynamics but with neural networks. SVG(∞) uses the model to calculate the gradient of real trajectories i.e. after each episode the Bellman equation is computed using as the transition probability, $p(S_{t+1}|S_t, A_t)$, the learned dynamics function.

2.4.3 Imagined data methods

Imagined data algorithms use the data obtained in the real environment to build a model than then will be used to obtain more data. Depending on the algorithm the data gathered in the real environment is also used to improve the policy.

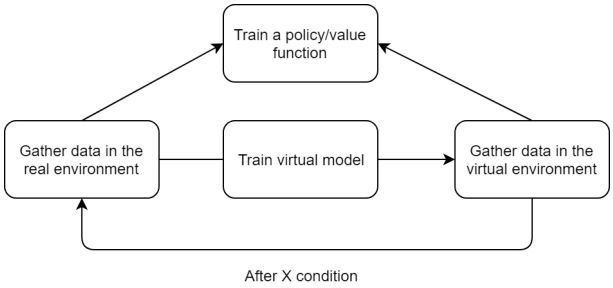


Figure 5: Imagined data Model Based Reinforcement Learning. Depending on the algorithm the arrow that train a policy/value function from the real environment is deleted.

In recent literature this kind of methods are often called Dyna-Q style algorithms. Dyna-Q [32] is perhaps one of the oldest model based algorithms. The basic idea is to combine table based Q-learning with learning a model. The process is the following: 1) gather a tuple (s, a, r, s') inside the real world, 2) create a model $f(s, a)$ and 3) obtain n tuples (s, a, r, s') from imagined trajectories. Both real and imagined tuples are used after each interaction to train Q-learning and the three steps are repeated until the desired performance is achieved. In [32] the imagined data helped table based Q-learning achieve faster convergence.

Recent work such as Model Ensemble Trust Region Policy (ME-TRPO) [19] Optimization uses an ensemble of models to fight against model bias and erroneous predictions with an on-policy algorithm. Model Based via Meta Policy Optimization (MB-MPO) [5] uses also an ensemble of models but an additional step of meta learning is used to obtain a policy robust to all models in the model ensemble. Two on-policy algorithms are used for MB-MPO. MB-MPO is based on the meta learning framework for Reinforcement Learning that learns a policy that is able to adapt to different tasks easly [8]. Both ME-TRPO and MB-MPO are used in MUJOCO tasks. In our case we use ME-TRPO as a starting

point to use off-policy algorithms due to its simplicity and robustness.

In the visual domain some new methods have been applied succesfully. In World Models [10] a three part system was used to complete a driving task from a videogame-like environment: 1) the visual part formed by a variational autoencoder [18] that reduced the dimensionality of the image input, 2) a memory formed by a long short term memory [16] used in predicting future states and 3) a controller (policy) formed by a 2 layer neural network trained using evolution strategies. In Model Based Reinforcement Learning for Atari [17] the results from DQN [23] on Atari environments are improved. The model of the environment is represented as a complex neural network (a mix of convolutional layers, feed forward layers and long short term memory) combined with an on-policy reinforcement algorithm, PPO to obtain the policy. This method improved the results from current model free methods on atari (DQN, Rainbow [14], PPO). The impressive results in World models and Model Based for Atari are achieved, in part, thanks to the use of latent representations of the dynamics (as images have an enormous space state) but as we will use MUJOCO tasks our state vector will be of the order of $10^1 - 10^2$ values and as consequence we do no need to use such methods. In addition to this, it could be that to predict the transition from an image to the next one (it may be that just a few pixels vary) may be easier than the prediction of complex robotic dynamics.

Model Based Reinforcement Learning is considered more sample efficient than Model Free Reinforcement Learning because it can use the model of the environment to sample enormous quantities of data and it can use the real data to both train the policy and train the model. Despite this, Model Based Reinforcement Learning still has not reached the performance of their Model Free counterparts.

2.4.4 Problems of World Models

The promise of Model Based Reinforcement learning is hindered by a set of problems that affect their potential performance.

The model, $f(s, a)$, is an approximation of the real environment. This makes that every time that you take an imagined step in this world model, an error is being introduced into the policy. If we use the model to predict trajectories over long horizons the error propagates and can get out of hand.

In addition, the error increases on the zones where you have no data of. Let's consider a model built by exploring an environment with a dataset gathered from a certain policy. If the policy is improved inside the world model and explores areas far apart from the dataset distribution of states, the error will increase greatly.

A problem related to the ideas mentioned is model bias. Kurutach et al., in [19], noticed that policies tend to concentrate on regions where less data is available leading to faulty policies. One of the possible reasons is that, in inaccurate regions, the rewards can be overestimated, leading to the policy ending in those fake optima. This issue is called

model bias or **model overfitting**.

Recently a benchmark for Model Based Reinforcement Learning algorithms [20] tested the different methods available. They found that virtual data methods often do not arrive at the same optima as their Model Free counterparts. This may be because the problems we have mentioned in this section. In particular SAC and TD3 improved over ME-TRPO and similar methods in most of the environments.

2.4.5 Model Ensemble Trust Region Policy Optimization

Model Ensemble Trust Region Policy Optimization tries to learn robusts models in order to avoid model bias.

1. First, an stable on-policy algorithm is used: Trust Region Policy Optimization. Trust Region Policy Optimization [28] makes sure that when the policy is updated the new policy is not far away from the old policy in the parameter space.

2. Second, virtual models compute Δs instead of s i.e. $f(s,a) \simeq \Delta s$. This is inspired by Nagabandi et al. [24] in which it is argued that calculating the difference between states leads to better results.

3. Third, instead of using one model, K models are used, where each model only differs from the others in the initial neural network weights and how the data is fed into the training procedure.

   When the policy executes an action inside the virtual world, a model from the ensemble is sampled randomly and the next Δs is computed using that model. This makes the process not to overfit to one specific model, avoiding the model bias problem.

   Each model is trained from a dataset D, gathered from the real environment. When training the dataset is divided in training and validation D_t, D_v and when the accuracy from the validation set does not increase after a certain number of epochs the learning is stopped.

4. Finally, validation is used when learning the policy inside the world model in order to know if our policy still improves in most of the models in the ensemble. The validation method is comparing the return of a few episodes from the last update policy against the current one in each of the models.

$$\Big[\frac{1}{K}\sum_{k=1}^{K}\mathbf{1}(G_{k,new} > G_{k,old})\Big] > Th \tag{42}$$

   If this inequality holds we continue training. This will allow us gather more real data when our policy is not improving in enough models. In other words, if there is only a few models where it is improving it is biased on those models, it should

improve on most of them hence we go back to the real world. Usually this threshold is kept between 0.3 and 0.7 depending on the environment.

ME-TRPO focuses on learning only the next state and not the reward in environments where the reward can be predicted directly as a function of the state. This is good for us as for HER we will be using sparse rewards and the reward will be the distance between goal and current state.

Algorithm 3 ME-TRPO [19]

Initialize a policy π_θ and all models $f_{\phi_1}, ..., f_{\phi_K}$.
Initialize a dataset D.
for each episode **do**
    Collect samples from the real environment with π_θ and add them to D.
    Separate dataset into training and validation D_t, D_v
    Train all models using D_t until the validation accuracy on D_v stops improving.
    while performance of policy in the models improves **do**
        Sample an initial state from saved initial states from the real environment, s.
        for 0,max_steps-1 **do**:
            Choose an action, a, according to the policy, π_θ.
            Act on a model chosen at random $[f_{\phi_k}]_{k=1}^{K}$ and observe s', r
            if terminal break.
            s=s'
        Every $\mathbf{C_1}$ steps: update policy using TRPO on the fictitious samples.
        Every $\mathbf{C_2}$ steps: validate the models comparing old and new policies.

This algorithm works well for MUJOCO tasks as it surpasses the sample efficiency of on-policy Model Free algorithms such as TRPO and even some off-policy such as DDPG in some environments. This algorithm will be our starting point when creating a off-policy model based system for sparse rewards.

Despite the good results reported, in Benchmarking Model Based [20], some late model free methods such as TD3 and SAC usually surpass ME-TRPO probably because they are off-policy that use experience replay and are more sample efficient than TRPO.

As we stated before one contribution of this work will be to combine off-policy methods with Model Based methods to have the best of before.

3 Integrating HER and Model RL

In this section we will explain how we have integrated Hindsight Experience Replay and Model Based Reinforcement Learning. Our idea is to find an algorithm a) works with sparse rewards and b) is sample efficient. Additionally, we would like to work in a robotic environment showing that this can have an applications in the real world.

For sparse reward we have chosen HER due to its efficiency on tackling problems of such kind.

To be sample efficient we will focus on Model Based Reinforcement Learning and off-policy methods because both are the ones that use samples more than one time to train. The mix of off-policy plus Model Based is, possibly, one of the ways in which a sample can be used the most.

For off-policy algorithms we have chosen SAC due to its stability compared to other off-policy algorithms. For Model Based Reinforcement Learning we have chosen ME-TRPO changing TRPO for SAC and becoming what we call **Model Ensemble Soft Actor Critic, ME-SAC**, due to its simplicity for implementation and its good results in robotic environments.

One important point to note is that even if we would had wanted to use an on-policy algorithm it could not have been possible due that HER needs an off-policy algorithm to work as it uses Experience Replay.

The prerequisites of implementing ME-SAC and ME-SAC with HER is to implement SAC with HER. To do this we just adapt the algorithm to HER following its pseudocode in (2) to be able to use SAC. To the best of our knowledge when we started this project there was no implementation that we could find about SAC with HER. Despite this, after some time we could find some implementation such as the one in [15].

In section 3.1 we will explain how we have implemented ME-TRPO with SAC. In 3.2 we will mix this implementation with HER.

3.1 Model Based Reinforcement Learning for off-policy algorithms

Model Based Reinforcement Learning with off-policy algorithms have yet to be explored according to Benchmarking Model-Based Reinforcement Learning [20]. We have implemented Model Ensemble with Soft Actor Critic in a straightforward way, where TRPO was used, it was changed for SAC. Albeit, some changes are considered:

- We will be updating online SAC i.e. ME-TRPO updates after recollecting 50000 samples. Instead, after every interaction with the environment, we will be updating SAC sampling a minibatch from the memory.

- As off-policy algorithms are more unstable than on-policy ones we have put in a rule set to control stability. After every iteration in the world model we do a simple check, consider π_{old} as the policy before updating in the world model and π_{new} the policy after sampling imagined data.

$$\pi_{new} = \begin{cases} \pi_{new} \text{ if } (G_{k,\pi_{new}} > G_{k,\pi_{old}}) \\ \pi_{old} \text{ otherwise} \end{cases} \tag{43}$$

  This is basically a check that the new policy after the world models is better than the old one in case of catastrophic forgetting.

- We try getting data from the virtual world only and also mixing virtual and real world data.

- In case of mixing real and virtual data the memory replay from off-policy methods will be divided in two. One for the imagined world and one for the real world.

One key difference between TRPO and SAC is that TRPO discards the data after being used while SAC does not. Due to that SAC may have data from previous models mixed with data from current models.

In addition an important hyperparameter that we will have to take into account is when to do the policy validation inside each model. ME-TRPO checks every 5 updates of 50000 virtual samples each but as off-policy are less stable and usually improve faster we expect for our models to enter to inaccurate predicted zones with ease. We will call this parameter, $\mathbf{C_2}$, **checkpoint**.

Apart from these points the implementation is the same as ME-TRPO in algorithm (3). The merge of SAC with ME-TRPO will be called ME-SAC as we will have a Model Ensemble with Soft Actor Critic. The full process is as following:

1. Gather data in the real world with SAC policy and put the samples inside a dataset, D. Train the policy with real world data if the mixing is enabled, this last step is not done in ME-TRPO.

2. Separate the dataset into training, D_t and validation, D_v. Train K models until the validation data accuracy does not increase after 5 consecutive steps. Every validation check is done after 5 epochs. The training is done to predict Δs and the reward is computed from the new state and action by a formula given by the environment. We have normalized the data with mean and standard deviation.

3. Gather data inside the virtual world. Update once per step with the desired batch size. If the policy does not improve in enough models after 5 checks return to the real world as in (42). The threshold of models is set to 0.3. For ME-TRPO the checks are done every 5 updates of 50000 interactions. In our case we will use this as an hyperparameter.

4. In addition if the final policy is not better than the old policy, return the old policy.
5. Repeat until good performance

Algorithm 4 ME-SAC with HER [1] [19]

Given: an off-policy RL algorithm $\mathbb{A}$, a strategy for sampling goals $\mathbb{S}$, and a reward function dependant on goals $r(s, a, g)$
Initialize a policy π_θ and all models $f_{\phi_1}, ..., f_{\phi_K}$. and a dataset D.
for each episode in the environment **do**
    Initialize s from the environment and desired goal g_d
    for each step in the episode **do**:
        $a = \pi(s||g_d)$
        Act on the environment and observe r,s'
        $s = s'$
        Store tuple (s,a,r,s') in the D for the model ensemble.
        if terminal break.
    if it is time to train models: **then**
        Separate D into training and validation D_t, D_v
        Train all models using D_t until the validation accuracy on D_v stops improving.
    while performance of policy in the models improves **do**
        Sample an initial state from the saved initial states from the real environment, s.
        for each step in the virtual episode **do**:
            $a = \pi(s||g_d)$.
            Act on a model chosen at random $[f_{\phi_k}]_{k=1}^{K}$ and observe r,s'
            s=s'
        for each step taken in the recent episode **do**:
            $r = r(s_t, a_t, g_d)$
            Store tuple $(s||g_d, a_t, r_t, s_t + 1||g_d)$ in replay memory, R
            Sample a set of additional goals from $\mathbb{S}$ for replay G.
            for $g' \in G$ **do**:
                $r' = r(s_t, a_t, g')$
                Store tuple $(s||g', a_t, r_t, s_t + 1||g')$ in replay memory, R
        for update = 1, N **do**
            Sample a minibatch from the replay buffer
            Perform a gradient descent step on the minibatch
        Every $\mathbf{C_2}$ steps: validate the models comparing old and new policies.

3.2 How to integrate HER with Model Based Reinforcement Learning

The final step of our project is to implement ME-SAC with HER. The implementation of ME-SAC with HER will follow closely ME-SAC except for that after each episode (in

the virtual world) we consider a new set of goals, G', sampled with strategy **S** (which will be future which is the one that has best performance). The memory of our agent will be filled with imagined goals in the virtual world. From this the agent will sample tuples $(s\|g, a, r, s'\|g)$ and update according to it.

We hope that the agent is able to learn to reach the objective with only imagined goals in the imagined environment.

The full algorithm can be found in (4).

3.3 Testing domains

For testing we have chosen 2 domains based on MUJOCO physics engine. Both these environments can be accesed using OpenAI gym library [3].

The first one is **HalfCheetah-v2**. The goal of this environment consists on make quadruped creature learn how to run as fast as possible. The state space consists of 17 continuous ($s \in \mathbb{R}$) features that are based on the joints of the creature plus their velocity status. The action space consist of 6 actions ($a \in \mathbb{R}$) which result in applying acceleration to the different joints of the creature. The reward is a sum of two parts: a control part that is computed as $-0.05 \cdot a^2$, where a is the vector of actions, which penalizes the use of high value actions (i.e. you do not want to use a lot of acceleration into the joints) and a velocity part that is computed as the velocity which rewards the half cheetah into moving as fast as possible.

The original Half Cheetah environment is not able to calculate the reward from the state as the velocity does not form part of the observations and ME-SAC is not designed to predict the reward in addition to the state, but to calculate the reward from the predicted state. To solve this we include an additional observation (making a total of 18 observations) that is the velocity to be able to calculate the reward from the state and only predict Δs (as in ME-TRPO) instead of Δs and r.

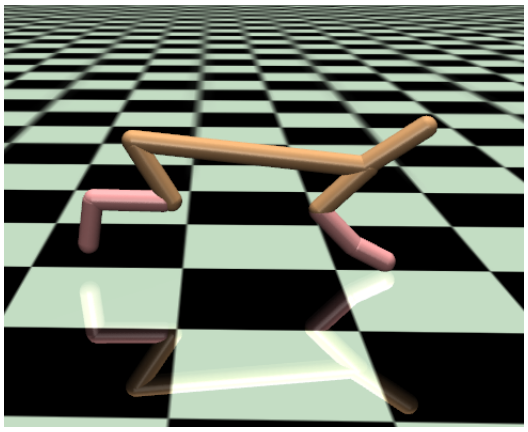


Figure 6: Modified Half Cheetah environment.

In this environment we will test ME-SAC. Then we will be able to extrapolate the results to another environment with sparse rewards.

The second one is **FetchReach-v1**. In this one the goal is to teach a robotic hand in 3 dimensions to grab an object. The state space is a vector of 10 dimensions ($s \in \mathbb{R}$) formed by the different joints of the robotic arm in each of the dimensions and its velocity plus the position of the objective. The objective is formed by a 3D cartesian vector. Actions are a 4 dimesion vector ($a \in \mathbb{R}$) and apply acceleration to the different joints of the arm. The reward is created in order to be sparse: it will only be 1 in the case that the distance from the objective is less than a certain threshold (0.05 simulated units). In this environment we will test if our implementation of SAC with HER works and if HER can work with World Models.

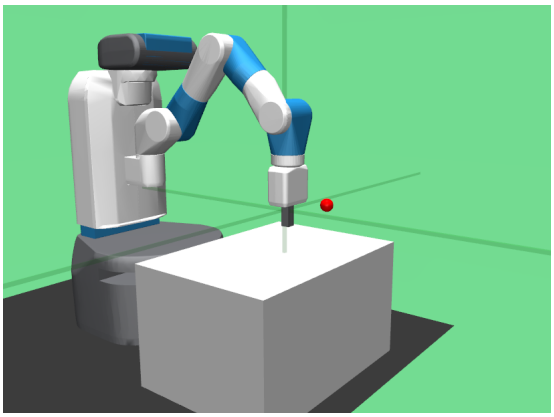


Figure 7: Fetch and Reach environment, the red dot is the goal.

We have chosen these environments because they are widely used in the literature. Specifically, Fetch and Reach (or its variants) for sparse rewards setting, is one of the few environments that are build for this problem. Unfortunately as MUJOCO environment require a license we were able to use only one computer to do the tests.

4 Results

In this section, we will showcase the results obtained in the two introduced environments, modified Half Cheetah and Fetch and Reach, for the three different algorithms we are trying to implement: SAC + HER, ME-SAC and ME-SAC + HER. To start off we will show a table with the different hyperparameters used for SAC.

β	0.1	**Number of layers**	2	**Policy**	Gaussian
α	0.0003	**Hidden size**	256 units	**Batch size**	256
γ	0.99	ρ	0.005	**Target update interval**	1

Table 1: SAC hyperparameters.

We use default hyperparameters for SAC with Half Cheetah found in [26]. As we can see the entropy parameter, β, is kept to a low value of 0.1. The entropy parameter could be automatically tuned but in our case we have decided to keep it constant as it has less computational complexity. The number of layers and hidden size is the same for all neural networks function approximators (the value function, the two action value functions and the policy). The target is updated one time per online update and is averaged using ρ via polyak averaging. The memory size is not mentioned because we will change it depending on the environment we are learning from.

For the experiments we used a licensed physics simulator, MUJOCO, with a student license. Unfortunately, as we only had one license we could not run the experiments on a server with high computational capabilities. Instead we have used a personal computer with a cpu i7-7500U 2.7 GHz with 12 GB DDR4 Memory. The experiments have been done in the CPU for reproducibility purposes as GPU results changed between runs.

We have used open source implementations in Pytorch for PPO [2] and SAC in [26].

Some of the presented figures have been smoothed with an exponential smoothing for clarity purposes. Consider the value of the result at an episode t given by x_t then the value shown in the figure is $v_t = sx_t - (1-s)x_{t-1}$ for $t > 0$ where s is a smoothing constant .

Finally, one key issue we have had during the whole research process is timing. As we only had one personal computer to experiment with we did not have as much time as we would have liked to produce results. This was specially notable during the tests of world models. Due to that, only two random seeds have been used in all the experiments.

4.1 SAC + HER

In this section, we will explain the results obtained with SAC with HER. As mentioned in section 3 we will be using Fetch and Reach to test the combination of SAC and HER. In this environment we have used a memory of size 10^6 which was the default used in the

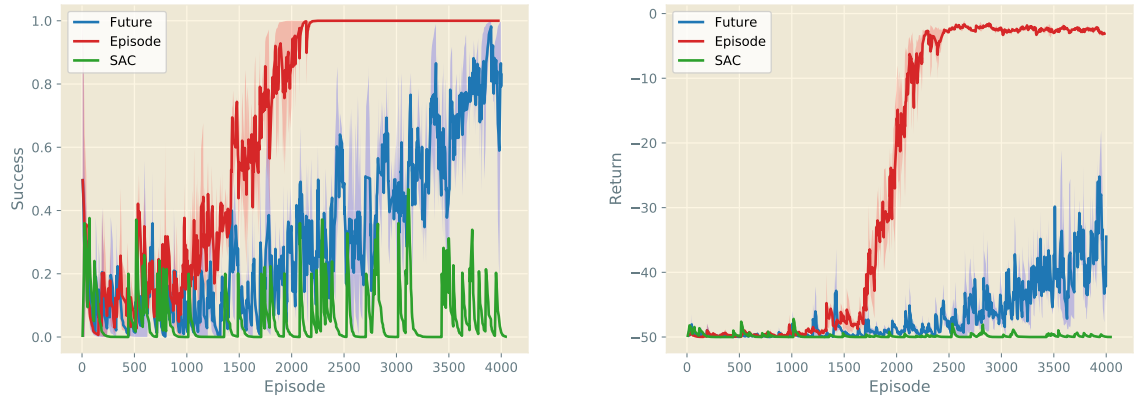
paper of HER [1].

In addition to SAC hyperparameters we need to set up the ones for HER. HER needs the number of extra goals sampled per episode, as well as the sampling strategy, $\mathcal{S}$, and the number of updates per episode, N (recall that in (2) after each episode we sample N minibatches from the experience replay and for each minibatch we update the network parameters once):

Number of extra goals	4
$\mathcal{S}$	future or episode
Number of updates per episode (N)	40

Table 2: HER hyperparameters. Number of extra goals and number of updates per steps are similar to the ones used in [1].

We have only tested the two best sampling strategies reported in [1] (Episode and Future). Additionally, we will be comparing the results with SAC without HER as a baseline. The results can be seen on the figure below:



(a) Rewards per episode. Smoothing of $s = 0.7$ used.

(b) Success per episode. Smoothing of $s = 0.2$ used.

Figure 8: Evolution of rewards and success per episode during training with SAC and SAC + HER with Future and Episode strategies. "Future" label refers to SAC + HER with Future goal sampling strategy, "Episode" to SAC + HER with Episode sampling strategy and "SAC" to standard SAC.

We can clearly see that SAC with HER with any of the sampling strategies is able to solve the environment (i.e. have a 100% success rate per episode) while SAC without HER is not able to see any kind of progress for around 4000 episodes. In addition, we can observe that Future sampling strategy outperforms Episode sampling strategy which

is the same conclusion as reported in [1] with DDPG.

While this environment is sparse compared to other commonly used, we can see that even SAC without HER is able to see meaningufull rewards in some episodes as the success is greater than 0 in some cases. This is true because the goal can be achieved with some luck randomly. In any case, the reward signals are so few that it does not learn even after 4000 episodes.

In addition, we will analyse the time complexity for each of the methods:

Model	**Time** (h min)
Episode	1h 43 min
Future	1h 54 min
SAC	1h 48 min

Table 3: Execution times per method to arrive to episode 4000.

All of the methods have similar time complexities, note that HER does not have affect performance.

4.2 ME-SAC

In this section, we test ME-SAC in the modified Half Cheetah environment. The idea is to treat Half Cheetah environment as a testing ground to make a learning algorithm that then can be applied in Fetch and Reach. In section 4.2.1, we will try virtual learning only, SAC learning in the virtual world. As the number of hyperparameters is enormous, we will focus on the most sensitive one: how much imagined data we use before every checkpoint (the parameter $\mathbf{C_2}$). In section 4.2.2, we will test if learning both in the real and virtual environments helps to improve the speed in learning. Then in section 4.2.3, we will compare it with an on-policy model based algorithm. Finally, in section 4.2.4, we will analyse the time complexities.

Regarding default SAC hyperparameters we have set memory size at 10000 as we see that it has a slight better performance for both SAC and ME-SAC and number of updates per step to the default 1.

The models from the ensemble predict Δs and consists of 2 layers of 1024 neurons each with Relu non-linearity. The dataset is formed by the trajectories sampled from the real environment and divided 66% into training set and 33% into validation set. The training is done with gradient descend with ADAM optimizer ($\alpha = 0.003$). The listed parameters for model ensemble are the same as in [19]. The remaining Model ensemble hyperparameters are listed below:

C_2	100, 200 and 2000 environment steps
Number of models	5
Number of checkpoints	5
Window size	30

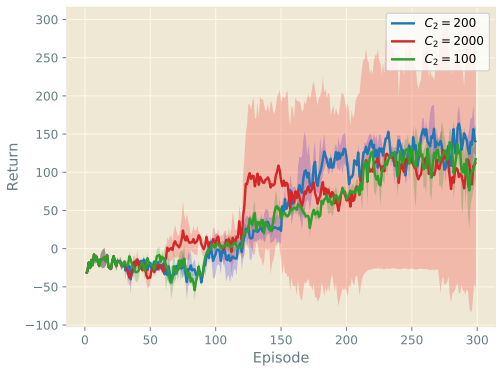
Table 4: ME-SAC hyperparameters.

Where number of models are the number of models used for the ensemble, number of checkpoints are the number of validations that the policy in the model ensemble is allowed to fail before it is forced to return to the real environment to gather more data and window size is the number of episodes the policy recollects data in the real world before training the model ensemble.

4.2.1 Virtual learning

In this section, we test if off-policy Model Based Reinforcement Learning algorithms can learn from imagined data.

The agent recollects data in the real world for 30 episodes (window size parameter) in outer iterations (iterations in the real world), then trains a model ensemble and data is recollected in them, in inner iterations (iterations in the virtual world). The data is recollected in the virtual world until the validation checkpoints fail more than 5 times. The full process is repeated until episode 300. The validation checkpoints are done every C_2 steps; we will vary C_2 to see the best performer. The results can be seen in the figure below:

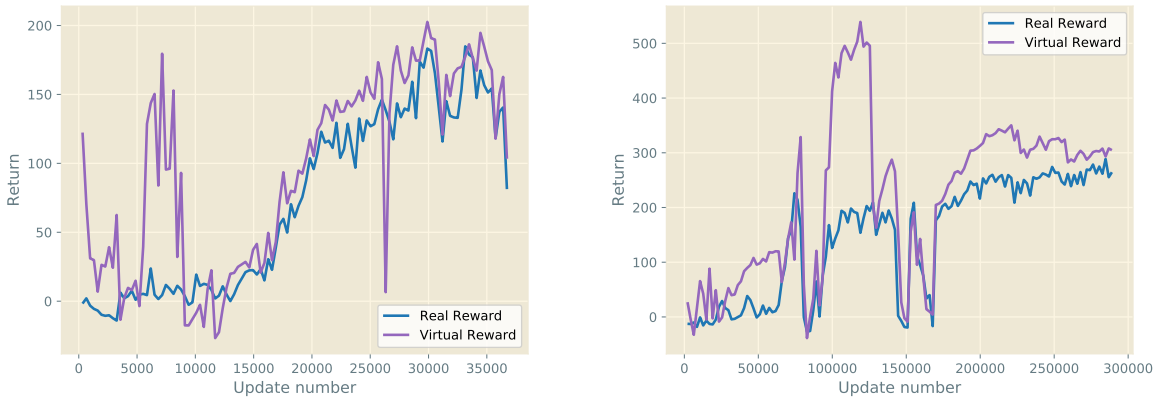


(a) Training return per episode. The return is smoothed with $s = 0.6$.

(b) Evaluation per episode. Evaluation is done without entropy term (i.e. $\beta = 0$). No smoothing is used.

Figure 9: Reward evolution for ME-SAC. The training return was computed after each episode. Evaluation return is obtained as an average over 100 evaluation episodes. Evaluation is done every 10 training episodes.

This figure shows that, in fact, **off-policy methods can learn with Model Based Reinforcement Learning**. As we can see, the mean result over 2 seeds are similar for the three C_2: 200, 2000 and 100 but the standard deviation is disparate. $C_2 = 2000$ outperforms all the others for one seed but fails to learn in another one (that is why it has a high standard deviation). Meanwhile the other two are more stable in comparison. This can be easily explained as an overfitting issue, $C_2 = 2000$ does not check policy performance as often as the other two and hence it is not as robust. This issue can be seen better if we compare the virtual learning vs real learning when doing the checkpoints in the virtual world:



(a) $C_2 = 200$. (b) $C_2 = 2000$

Figure 10: Real vs Virtual learning comparison for one of the seeds used. In the checkpoint evaluation, the return is computed every C_2 updates. It is the average of 100 evaluation episodes and it is computed for both the virtual world and the real world. No smoothing is used.

To start we can observe that, the total number of updates for $C_2 = 2000$ is nearly 10 times the number of updates for $C_2 = 200$. In the case of the baseline it would have around 30000 number of updates (as we have 300 episodes of duration 100 steps each and it updates once per step) which is similar to $C_2 = 200$ but much less than $C_2 = 2000$.

The idea of world models is that we could obtain orders of magnitude more virtual data compared to Model Free algorithms in order to outperform them but it seems that, in the case of $C_2 = 200$, the policy is not able to surpass the checkpoints to stay enough time to gather much more data than the baseline. Instead, for $C_2 = 2000$, it is not a problem.

For this seed, $C_2 = 2000$, is able to outperform $C_2 = 200$ as seen in the figure above.

For $C_2 = 200$ both real and virtual rewards follow a similar trend except in two points: a) the beginning where the agent may not have enough information to build a robust en-

semble of models and b) at a singular peak near update 25000 which would make our policy to unlearn but thanks to the policy check after each world model iteration we avoid it.

In the case of $C_2 = 2000$ the discrepancies between real and virtual returns increase both in value and times that a discrepancy appear. Our hypothesis is that if you have enough discrepancies it may lead to not learning a correct policy which is what happens for $C_2 = 2000$ in the case of the second seed.

Next we will focus on the loss evolution when training the model ensemble with supervised learning.

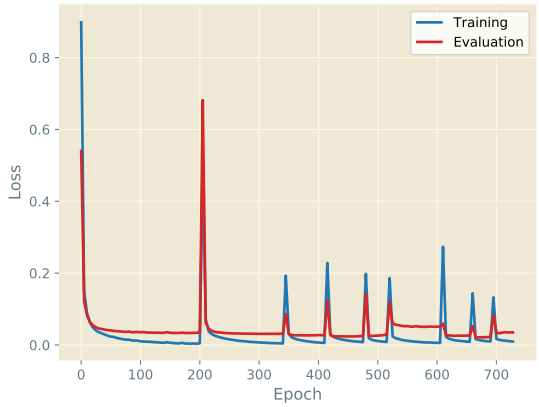
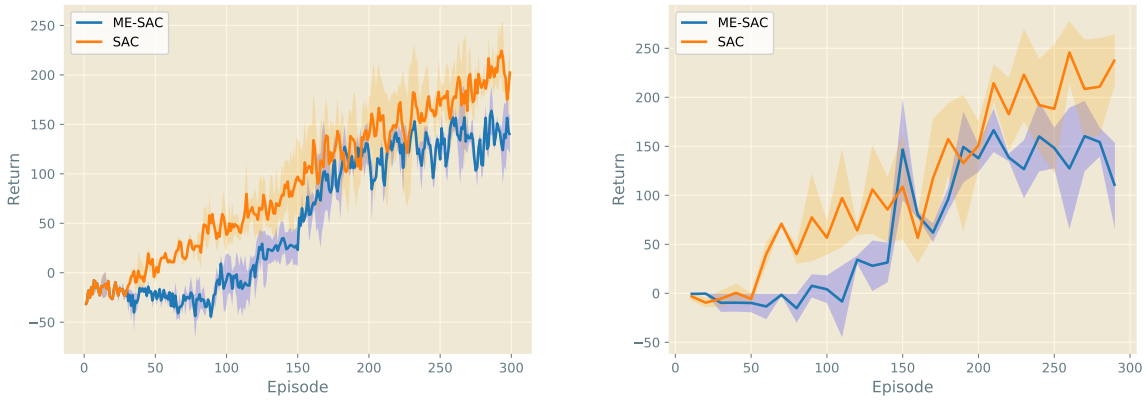


Figure 11: Training and evaluation loss evolution over the training epochs for a single model in the ensemble for the case of $C_2 = 200$.

Figure 11 shows the evolution of train and evaluation loss for one of the models used in the model ensemble. The figure shows a periodic evolution of starting with a high loss and then a decrease. The meaning of this periodic evolution is: initially the model is trained until the validation data stops increasing, then, after another inner and outer iterations, the model is trained again resetting the ADAM optimizer and with new obtained data. With the newly obtained data the loss increases again and it is trained once more until the validation loss stops decreasing.

In the figure, we can count up to 9 peaks meaning that the model has been trained 9 times, which agrees with that we are doing outer iterations of 30 episodes and a maximum of 300 episodes.

Finally we can compare the value of the best ME-SAC performer ($C_2 = 200$ is slightly better than $C_2 = 100$) against the baseline SAC:



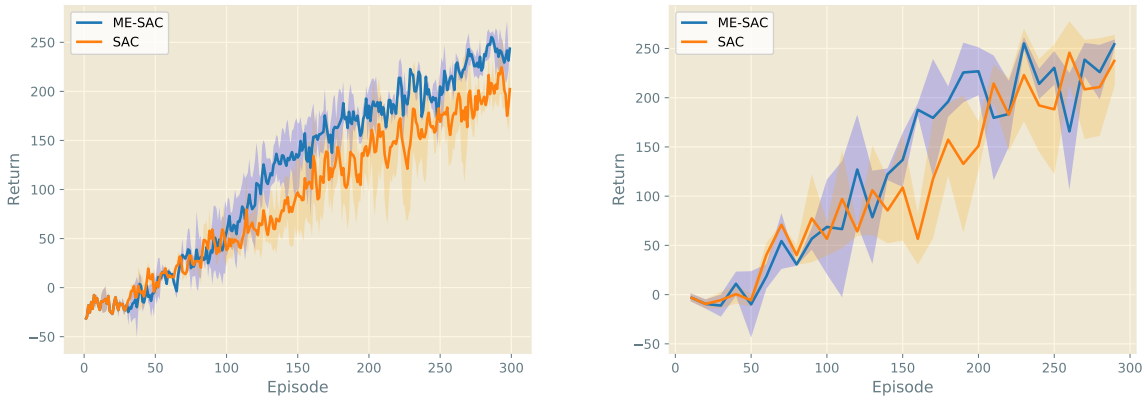
(a) Training. Smoothing with $s = 0.7$ is used. (b) Evaluation. No smoothing is used.

Figure 12: Comparison of SAC vs ME-SAC with virtual data. $C_2 = 200$

Unfortunately, the best performer over the parameters C_2 is not enough to outperform SAC. In any case, ME-SAC is able to learn in similar speeds compared to SAC.

4.2.2 Virtual and Real learning

As seen in the previous section, virtual learning only does not seem to surpass or even be equal to SAC in training speed. However, we can combine both real and virtual training to see if the combination is enough to exceed the original SAC. The result with $C_2 = 200$ training in both real and virtual worlds can be seen below:



(a) Training return per episode. Smoothed with $s = 0.6$. (b) Evaluation of SAC and ME-SAC. No exponenthial smoothing used.

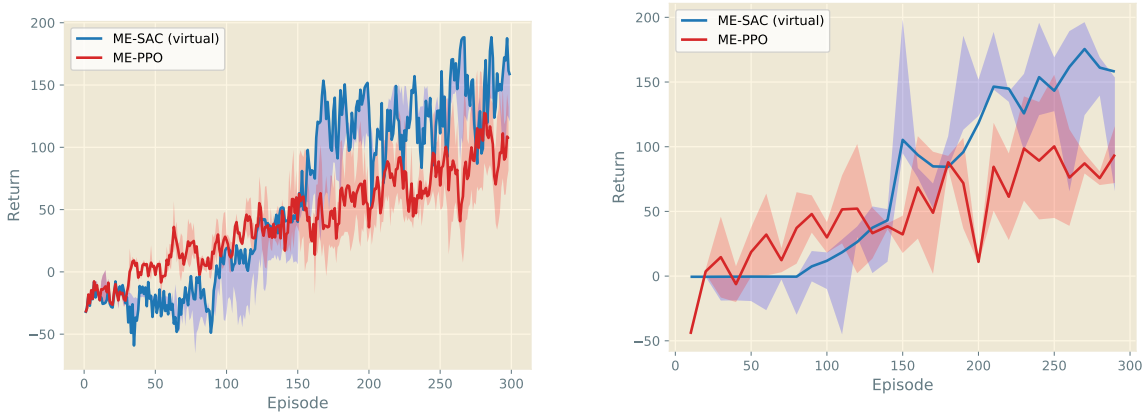
Figure 13: Comparison of returns between SAC and ME-SAC training in the real and virtual worlds.

Both methods achieve similar performances. Even though ME-SAC achieves better training return faster, the evaluation return is almost equal.

4.2.3 Comparison with ME-PPO

In this section, we compare ME-SAC against ME-PPO. The idea was to compare our method with ME-TRPO but PPO is a simpler substitute for TRPO that obtains similar (and usually even better) returns as reported in [29]. In addition, this comparison may serve as a basis of comparing off-policy against on-policy algorithms in world models.

We configure PPO with the default hyperparameters reported on [2] except for the batch size for learning which we use 50000 as in the paper of ME-TRPO [19]. Both results have been compared using the virtual world only. The results can be found below:

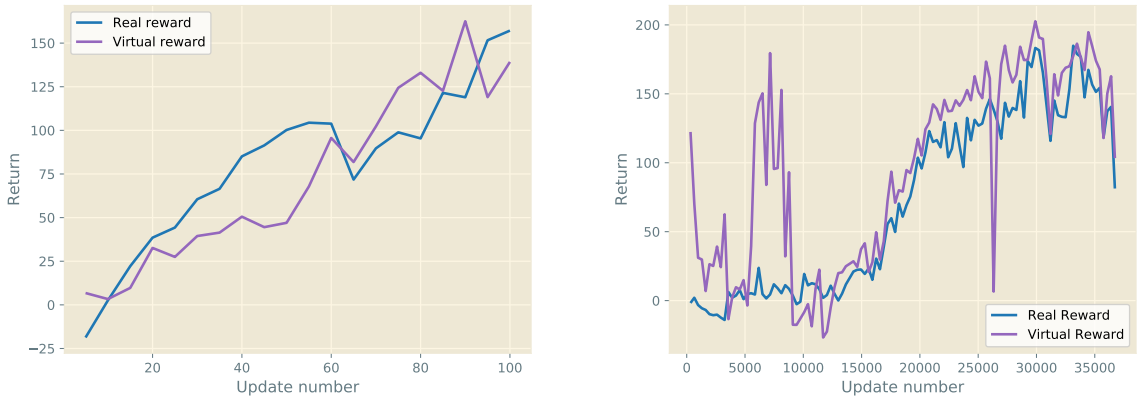


(a) Training of ME-PPO and ME-SAC. $s = 0.6$

(b) Evaluation of ME-PPO and ME-SAC. No smoothing used.

Figure 14: Comparison of ME-PPO and ME-SAC for $C = 200$.

While ME-PPO starts faster than ME-SAC, ME-SAC is able to surpass ME-PPO performance after some episodes. Interestingly ME-PPO appear to learn steadily from each world model iteration as there are no big increases in return after a small period of time. Instead, ME-SAC does have such increases for example near episode 150. Additionally, at some other points (the first 100 episodes) it appears to not be able to learn correctly. This may point that ME-SAC is less stable than ME-PPO. We can also compare the virtual learning of ME-PPO against the virtual learning of ME-SAC in order to see the stability:



(a) ME-PPO virtual learning. No smoothing is used.

(b) ME-SAC virtual learning for $C = 200$. No smoothing is used.

Figure 15: Comparison of virtual learning of ME-SAC and ME-PPO for one of the seeds used. ME-PPO checkpoints are done every 5 updates of 50000 steps as in ME-TRPO.

As the number of updates and batch size differs greatly it is not possible to compare them on the same figure. It appears that ME-PPO learns in a more stable way as in none of the checkpoints ME-PPO has catastrophic failures or great overestimations as in ME-SAC. This may be one of the reasons why ME-SAC is not able to learn faster.

4.2.4 Time complexity

In this section, we analyse the time complexity to complete the experiments for ME-SAC and compare it to SAC and ME-PPO. We expect higher training times for ME-SAC and ME-PPO due to the model computations.

Model	**Time** (h min)
ME-SAC ($C_2 = 2000$)	4h 8 min
ME-SAC ($C_2 = 200$)	2h 9 min
ME-SAC (Virtual and Real $C_2 = 200$)	2h 16 min
ME-PPO	6h 24 min
SAC	14 min

Table 5: Running times per method to arrive to episode 300. Reported times correspond to the average execution time of two seeds.

As we can see ME-SAC has much bigger training times than SAC, due to the training of the models, and the gathering of additional data inside the models. The mixing of virtual and real only increases the times in a few minutes. If we compare ME-PPO to

ME-SAC, ME-PPO needs much more time to obtain similar performance.

4.2.5 Conclusion

In this section, we have seen that **off-policy model based learning with imagined data is possible**. Despite this, the results show that ME-SAC can only achieve similar performance to SAC and not surpass it. We hope that in a simpler dynamics environment (for example Fetch and Reach), ME-SAC is able to learn better models which result in better imagined data to learn a better policy.

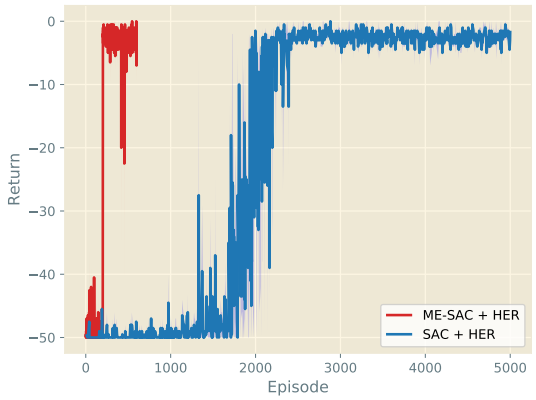
In addition, we have seen that ME-SAC is able to give a slight increase in performance to SAC if we use both real and fake data. Finally, ME-SAC achieved a better performance than ME-PPO but with higher instability.

4.3 ME-SAC + HER

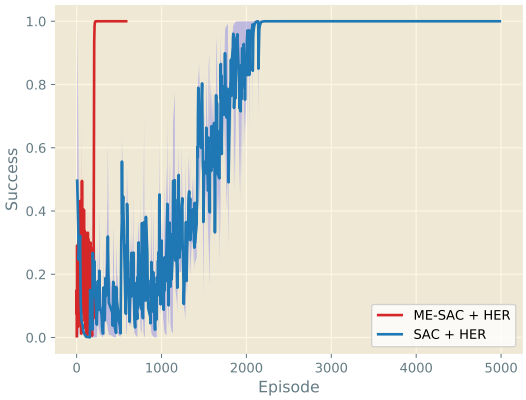
In this section, we will test ME-SAC + HER. The hyperparameters for SAC used in this environment are the same as in section 4.1 and the hyperparameters for the model ensemble are the same as in section 4.2 except for C_2 and the number of episodes per outer loop (window size parameter). To test ME-SAC + HER we will come back to the environment of Fetch and Reach and compare it with HER.

4.3.1 Virtual learning

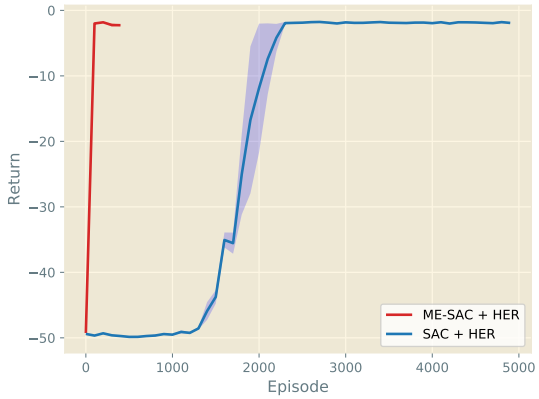
As the environment is different to modified Half Cheetah we do not know which C_2 parameter and how many iterations set for the outer loops. As the training takes much more iterations (as seen in section 4.1) we have set to 200 the number of episodes to recollect data (10000 steps) and we have tested different values for the checkpoint parameter C_2, $C_2 = [5, 30, 50, 200]$, where in this case C_2 is given by episodes instead of steps and we have found that 200 is the best overall. The results of ME-SAC with $C_2 = 200$ can be see below:



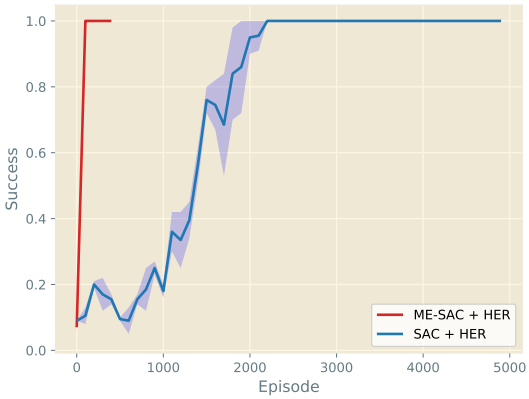
(a) Training return evolution over time. No exponential smoothing is used.



(b) Success evolution over time. Exponential smoothing with $s = 0.7$ is used.



(c) Return evaluation evolution over time. No exponential smoothing is used.



(d) Succes evaluation evolution over time. No exponential smoothing is used.

Figure 16: Results ME-SAC with HER. The results are averaged over two seeds. ME-SAC is run until episode 600 thus 2 outer iterations are produced but the environment is solved at around episode 200 in both seeds.

ME-SAC is able to solve this environment only with one iteration of the world model[5]. The great improvement over Half Cheetah may have various reasons:

1. Half cheetah environment is more complicated in the part of prediction from the model than Fetch and Reach. In Half cheetah there are 18 features and 6 actions while in Fetch and Reach there are only 10 features (3 of which are constant) and 4 actions, thus being able to predict more easily. In addition, the modified Half Cheetah has an horizon (maximum number of steps) of 100 timesteps while Fetch and Reach has an horizon of 50 timesteps and as the error propagates its much more easy the prediction in a 50 horizon environment. Finally, the undergoing dynamics may have a more complicated function in Half Cheetah.

[5]In some seeds it may take more than one iteration of the world model to converge.

2. HER environment difficulty may be in the sparse reward setting not in the problem per se and being able to see more data even if it does not have the quality of the real data may improve the speed for optimality.

Following the plot of the performance, we have plotted how does the loss evolves in the case of one of the models in the ensemble.

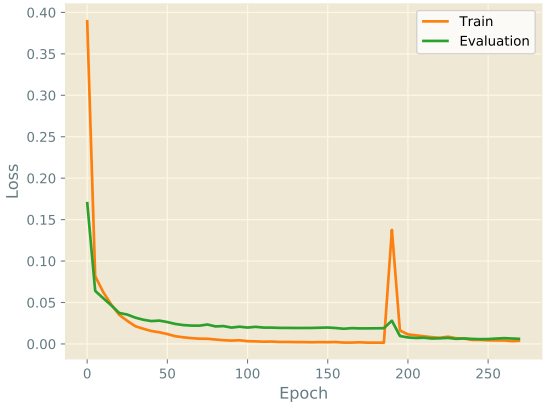


Figure 17: Training and evaluation loss over the training epochs on one of the models of ME-SAC with HER.

The loss follows a clear downwards trend and a peak is produced when new data is gathered as expected. As we can see, the model is trained only two times. In any case, the performance plot in figure 16, shows us that the agent only requires of one model to solve the environment, the second one is trained after that the environment is already solved.

Next, the evolution of real and virtual returns on the checkpoints is examined:

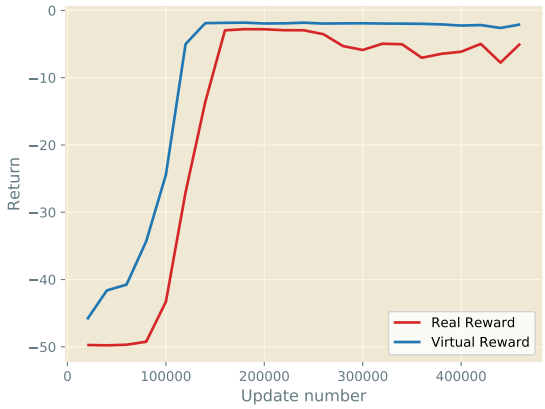


Figure 18: Virtual and real return comparison in the checkpoints. No exponential smoothing is used.

The prediction is not exact but it follows the same trend except on a few points. This supports our idea that the model is easier in this environment as the reward evolution is smoother than in the case of Half Cheetah in figure 10. One interesting point of discussion is the fact that the virtual model ensemble would count all points as a success because they have a reward less than 50 meaning that it achieved the objective in less than 50 timesteps.

Moreover, it appears that the optimal policy is achieved with $1.5 \cdot 10^5$ updates. Instead, SAC with HER arrives to the optimum with 2200 episodes of 40 updates each i.e. $0.9 \cdot 10^4$ updates, around 1 order of magnitude less. In other words it takes more virtual updates for the world model to arrive to the same point.

In this case, we accomplish the correct use of world models, we get an order of magnitude more of virtual data compared of what we could get with real data.

Finally, we will show, as mentioned at the start, the comparison of different values of C_2. Remember that in this case, C_2 is counted in episodes.

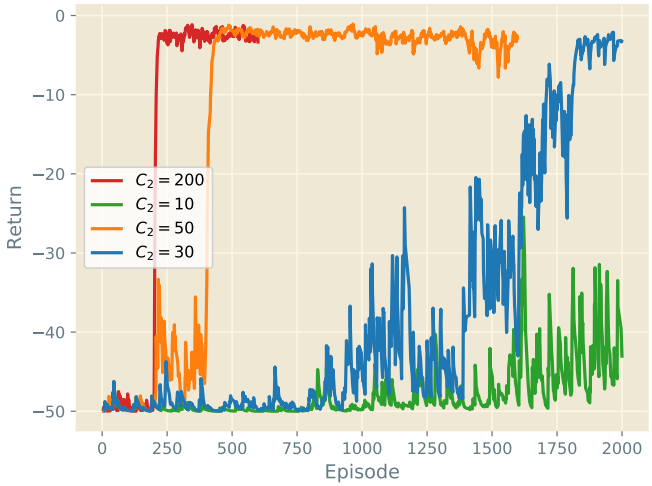


Figure 19: Comparison of different values of C_2 for ME-SAC with HER.

It appears that the best C_2 is the highest, 200 episodes. While in other environments increasing it tenfold may lead to instability in this case it works. As you reduce C_2 the performance is reduced.

As we had Half Cheetah as an starting point, initially we used much lower C_2, and we were not able to solve the environment without needing many episodes (compare the 200 episodes of 50 timesteps each (i.e. 10000 steps) we use in this case against the 200 steps in Half Cheetah). However, the environments differ greatly, thus, this parameter may not be adequate to transfer.

4.3.2 Time complexity

In this section, we have compared the times of execution between ME-SAC with HER and SAC with HER.

Model	**Time** (h min)
ME-SAC with HER (250 episodes) (virtual and real)	1 h 29 min
SAC with HER (250 episodes)	5 min
SAC with HER (2500 episodes)	1h 1min

Table 6: Time comparison of ME-SAC with HER to arrive to its optimum, around 250 episodes against SAC to arrive to the same result in 2500 episodes. We have also included the time it takes for SAC to arrive to 250 episodes despite it has no success yet for more information.

As we can see, ME-SAC with HER arrives to the optimum only with 1 h and 30 min, 30 minutes more than the baseline. In real world systems this favours ME-SAC as the bottleneck is in sample gathering.

4.3.3 Conclusion

In this section, we have tested if ME-SAC with HER is able to work with Fetch and Reach and compared the checkpoint hyperparameters where we found that $C_2 = 200$ is the optimal one.

Not only ME-SAC with HER is able to learn in Fetch and Reach but it achieves speeds **ten times faster than SAC for optimality**.

5 Conclusions

In this master thesis, we have investigated the promising idea of Model Based Reinforcement Learning and combined it with a method to deal with sparse rewards.

Our findings may motivate others in the investigation of a) imagined data Model Based with off-policy Model Free methods and b) the combination of HER and world models.

5.1 Contributions

1. Implementation of SAC with HER. To our own knowledge there was no public implementation when we started working on this project. Despite this there has been some implementations recently such as [15].

2. To the best of our knowledge, the investigation of off-policy Model Based methods with imagined data have been the first of its kind (according to [20]). The implementation of ME-SAC shows that off-policy algorithm may learn with synthetic data.

3. Showing that ME-SAC has slightly better performance than ME-PPO and it is not able to surpass normal SAC.

4. As far as we know, the investigation of world models with Hindsight Experience Replay is novel. We have demonstrated that ME-SAC together with HER is 10 times faster than SAC plus HER. This may lead to work around world models and sparse rewards.

5. The check used comparing old and new policies after each world model iteration was novel at the time of working with this thesis. Just a few days before delivering this thesis, there has been a similar idea but in the real environment [6].

5.2 Future Work

Regarding the results showing that ME-SAC is not able to outperform SAC in modified Half Cheetah, future work on improving ME-SAC can be investigated:

1. Use local models (different neural networks for different parts of the trajectory) instead of a single global model that learns the whole dynamics. This should have better precision toward the end of the trajectory.

2. Inspired by [17], a possible improvement may be start training (in the world model) not in the starting point of the environment but treating a sample of the experience replay of the real world as the starting point. This could help with environments with long horizons.

3. Investigate to improve the stability of off-policy algorithms inside the world model.

Regarding Model Based Reinforcement Learning and Hindsight Experience Replay, we need to test if we can replicate those results in other more difficult environments. As seen by the results in Half Cheetah we may need to finetune the world model part if the environments are complicated.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, OpenAI Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. In *Advances in Neural Information Processing Systems*, pages 5048–5058, 2017.

[2] Nikhil Barhate. Proximal policy optimization on pytorch. https://github.com/nikhilbarhate99/PPO-PyTorch, 2019.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[4] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. In *Advances in Neural Information Processing Systems*, pages 4754–4765, 2018.

[5] Ignasi Clavera, Jonas Rothfuss, John Schulman, Yasuhiro Fujita, Tamim Asfour, and Pieter Abbeel. Model-based reinforcement learning via meta-policy optimization. *arXiv preprint arXiv:1809.05214*, 2018.

[6] Vibhavari Dasagi, Jake Bruce, Thierry Peynot, and Jürgen Leitner. Ctrl-z: Recovering from instability in reinforcement learning. *arXiv preprint arXiv:1910.03732*, 2019.

[7] V Feinberg, A Wan, I Stoica, MI Jordan, JE Gonzalez, and S Levine. Model-based value expansion for efficient model-free reinforcement learning. In *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, 2018.

[8] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1126–1135. JMLR. org, 2017.

[9] Scott Fujimoto, Herke van Hoof, and David Meger. Addressing function approximation error in actor-critic methods. *arXiv preprint arXiv:1802.09477*, 2018.

[10] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

[11] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[12] Danijar Hafner, Timothy Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551*, 2018.

[13] Nicolas Heess, Gregory Wayne, David Silver, Timothy Lillicrap, Tom Erez, and Yuval Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pages 2944–2952, 2015.

[14] Matteo Hessel, Joseph Modayil, Hado Van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[15] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[16] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[17] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

[18] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[19] Thanard Kurutach, Ignasi Clavera, Yan Duan, Aviv Tamar, and Pieter Abbeel. Model-ensemble trust-region policy optimization. *arXiv preprint arXiv:1802.10592*, 2018.

[20] Eric Langlois, Shunshi Zhang, Guodong Zhang, Pieter Abbeel, and Jimmy Ba. Benchmarking model-based reinforcement learning. *arXiv preprint arXiv:1907.02057*, 2019.

[21] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[22] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[24] Anusha Nagabandi, Gregory Kahn, Ronald S Fearing, and Sergey Levine. Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7559–7566. IEEE, 2018.

[25] OpenAI. Pendulum environment. https://gym.openai.com/envs/Pendulum-v0/.

[26] Pranz24. Pytorch implementation of soft actor critic. https://github.com/pranz24/pytorch-soft-actor-critic, 2018.

[27] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In *International Conference on Machine Learning*, pages 1312–1320, 2015.

[28] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.

[29] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[30] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[31] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.

[32] Richard S Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning Proceedings 1990*, pages 216–224. Elsevier, 1990.

[33] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[34] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033. IEEE, 2012.

[35] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[36] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.